- Introduction
  - Getting Started
  - Hand-In Procedure
  - Synchronizing Access to Shared Objects
  - Test the User-Process Setup
- Part 1: Multicore Support
  - Per-CPU State and Initialization
  - Locking
- Part 2: Preemptive Multitasking
  - Clock Interrupts and Preemption
- Part 3: Preempting Kernel Execution
- Part 4: The Producer-Consumer Problem

CS422/522 Lab 4: Multicore and Preemption

due 2015-11-06

Introduction

This lab is split into four parts. In the first part of this lab, you will add multiprocessor support to mCertiKOS. Next, you will implement a preemptive scheduler, and make some designated parts of the kernel preemtable by turning on the interrupts during those parts of kernel code. Last, you will be asked to design and implement the producer-consumer problem (also known as the bounded-buffer problem) with shared objects and condition variables. Unlike previous lab, you are allowed to make arbitrary changes to the kernel source, including adding new source files, modifying any existing files, and removing existing files if necessary.

* Debugging may take much more time in the concurrent setting. START EARLY!

Getting started

In this and future labs you will progressively build up your kernel. We will also provide you with some additional source. To fetch that source, use Git to commit changes you've made since handing in lab 3 (if any), fetch the latest version of the course repository. Then create a local branch called `lab4` based on our lab4 branch, `origin/lab4`. Note that for this assignment, you SHOULD NOT merge the lab4 branch with your previous lab3 branch. Since adding the multiprocessor support requires many changes to the existing code, we have provided all the code you need to get started.

```
$ cd ~/cpsc422/mcertikos
$ /c/cs422/apps/syncrepo.sh
remote: Counting objects: 130, done.
remote: Compressing objects: 100% (70/70), done.
remote: Total 110 (delta 39), reused 110 (delta 39)
Receiving objects: 100% (110/110), 37.54 KiB | 0 bytes/s, done.
Resolving deltas: 100% (39/39), completed with 17 local objects.
From /c/cs422/repo/mcertikos
 * [new branch]      lab4       -> lab4
****
Remote repository synchronized successfully.
****
$ git pull
remote: Counting objects: 130, done.
remote: Compressing objects: 100% (70/70), done.
remote: Total 110 (delta 39), reused 110 (delta 39)
Receiving objects: 100% (110/110), 37.54 KiB | 0 bytes/s, done.
Resolving deltas: 100% (39/39), completed with 17 local objects.
```

```
From /c/cs422/SUBMIT/lab/xw247
 * [new branch]      lab4       -> origin/lab4
Already up-to-date.
$ git checkout -b lab4 origin/lab4
Branch lab4 set up to track remote branch refs/remotes/origin/lab4.
Switched to a new branch "lab4"
$
```

Hand-In Procedure

Include the following information in the file called `README` in the `mcertikos` directory: who you have worked with; whether you coded this assignment together, and if not, who worked on which part; and brief description of what you have implemented; and anything else you would like us to know. When you are ready to hand in your lab code, add the new files to git, commit your changes, and then run `git push` in the `mcertikos`.

```
$ git commit -am "finished lab4"
[lab4 a823de9] finished lab4
 4 files changed, 87 insertions(+), 10 deletions(-)
$ git push
```

Read Chapter 5: Synchronizing Access to Shared Objects

> **Exercise 1**
>
> If you have not had chance to read the Chapter 5 of the textbook "Synchronizing Access to Shared Objects", this is the perfect time to do so. Make sure you read the chapter carefully and follow the various guidelines presented in the book when you work on this assignment.

Test the User-Process Setup

In order to facilitate testing implementations of various topics introduced in this assignment (e.g., multicore, preemption in both kernel and user), we have developed an initial test setup for you. If you checkout `user/pingpong/ping.c` and `user/pingpong/pong.c`, you will see that we have wrote two user programs that call two system calls `produce` and `consume` at different speeds. These system calls, defined in `kern/trap/TSyscall/TSyscall.c`, simply print out 5 numbers with a loop. In `kern/init/init.c`, we created two producer processes on CPU #1, and two consumer processes on CPU #2.

With this setup, you will see different levels of message interleavings as you implement more and more parts in this assignment. For example, once you have implemented the first part (multicore support), you will see that the producer outputs and consumer outputs can interleave arbitrarily. However, you will notice that only one user process on each processor has ever been started, and if you just look at outputs from one particular CPU, they are perfectly ordered, as shown in a sample output below.

*As in the previous assignments, the provided program will not run until you implement required modules.

```
CPU1: process ping1 1 is created.
CPU1: process ping2 3 is created.
CPU2: process pong1 2 is created.
From cpu 1: ping started.
CPU2: process pong2 4 is created.
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 0
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 1
From cpu 2: pong started.
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 2: Consumed 0
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 2
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 3
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 4
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 0
```

```
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 1
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 2: Consumed 1
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 2: Consumed 2
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 2: Consumed 3
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 2: Consumed 4
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 2
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 3
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 4
...
```

You may have already guessed why the other process for each processor does not get any chance to run. If you checkout `user/lib/entry.S`, now we no longer wrap each user program with infinite sequences of yield calls. Instead, every process just spins once its main function returns. With the current cooperative multitasking scheme, the first process on each processor will never release the CPU to the other process since there is no explicit call to the function `yield`. To fix this issue, we need a preemtive scheduler, which we ask you to implement in the part 2 of this assignment. After part 2, you will see the outputs from both user processes on each CPU, and the set of outputs from each process may alternate, meaning the processes get preempted during their executions.

```
...
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 0
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 1
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 2
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 3
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 4
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 2: Consumed 0
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 2: Consumed 1
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 2: Consumed 2
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 2: Consumed 3
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 2: Consumed 4
From cpu 1: ping started.
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 4: Produced 0
```

```
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 4: Produced 1
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 4: Produced 2
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 4: Produced 3
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 4: Produced 4
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 4: Produced 0
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 4: Produced 1
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 4: Produced 2
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 4: Produced 3
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 4: Produced 4
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 4: Produced 0
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 4: Produced 1
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 4: Produced 2
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 4: Produced 3
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 4: Produced 4
From cpu 2: pong started.
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 3: Consumed 0
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 3: Consumed 1
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 3: Consumed 2
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 3: Consumed 3
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 3: Consumed 4
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 0
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 1
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 2
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 3
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 4
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 0
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 1: Produced 1
...
```

So, this is much better. On the other hand, if you check the individual process's output, the printed numbers from 0 to 4 are always consecutive, which means this part of the code was never preempted. This is because we always disable interrupt when the user traps into the kernel (check the first line of the implementation of the assembly function `_alltraps` in `kern/dev/idt.S`). Since the interrupt was always turned off inside the kernel, when the system call handlers (e.g., `sys_produce` or `sys_consume`) print out numbers 0-4 in a loop, the timer interrupt would not occur, thus they could not be preempted. In part 3, you will be asked to enable interrupt in system call handlers for the producer and consumer. After part 3, you should see that all the printed numbers are randomly interleaved as expected:

```
CPU1: process ping1 2 is created.
CPU2: process pong1 1 is created.
CPU2: process pong2 4 is created.
CPU1: process ping2 3 is created.
From cpu 2: pong started.
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 1: Consumed 0
From cpu 1: ping started.
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 1: Consumed 1
From cpu 2: pong started.
From cpu 1: ping started.
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 4: Consumed 0
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 4: Consumed 1
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 4: Consumed 2
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 4: Consumed 3
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 3: Produced 0
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 3: Produced 1
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 2: Produced 0
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 2: Produced 1
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 1: Consumed 2
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 1: Consumed 3
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 1: Consumed 4
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 1: Consumed 0
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 1: Consumed 1
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 4: Consumed 4
```

```
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 4: Consumed 0
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 4: Consumed 1
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 2: Produced 2
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 2: Produced 3
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 2: Produced 4
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 4: Consumed 2
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 4: Consumed 3
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 1: Consumed 2
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 1: Consumed 3
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 1: Consumed 4
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 1: Consumed 0
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 3: Produced 2
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 3: Produced 3
[D] kern/trap/TSyscall/TSyscall.c:147: CPU 1: Process 3: Produced 4
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 1: Consumed 1
[D] kern/trap/TSyscall/TSyscall.c:158: CPU 2: Process 4: Consumed 4
...
```

We have developed this simple test setup so that you can test your implementations in the first three parts before you dive into the actual implementations of producer-consumer in part 4. If you start directly on part 4 without fully testing the first three parts step by step, the debuging could become a nightmare because it may not be clear at all where the bugs are located. In part 4, you can freely design your kernel structure. You can add/modify/delete directories/files as you wish, but make sure that you add all the new files to the git repository by the `git add` command before you commit or push.

We do not provide any test scripts for this part of the assignment, because: writing a comprehensive test cases for many parts of this assignment would leak too much information on the required implementation; or some parts of the code are very difficult to test. For many other parts, you should now be quite familiar on how to write good test scripts for them, so we no longer provide simple scripts to get you started as in previous assignments. On the other hand, we encourage you to write your own test scripts to challenge your implementation. The more of your own code is covered by the test cases, the less likely that any bugs will be found during the actual grading.

Part 1: Multicore Support

We are going to make mCertiKOS support "symmetric multiprocessing" (SMP), a multiprocessor model in which all CPUs have equivalent access to system resources such as memory and I/O buses. While all CPUs are functionally identical in SMP, during the boot process they can be classified into two types: the bootstrap processor (BSP) is responsible for initializing the system and for booting the operating system; and the application processors (APs) are activated by the BSP only after the operating system is up and running. Which processor is the BSP is determined by the hardware and the BIOS. Up to this point, all your existing mCertiKOS code has been running on the BSP.

In an SMP system, each CPU has an accompanying local APIC (LAPIC) unit. The LAPIC units are responsible for delivering interrupts throughout the system. The LAPIC also provides its connected CPU with a unique identifier. An implementation of the LAPIC driver can be found in `kern/dev/lapic.h` and `kern/dev/lapic.c`. It is ok if you do not understand the code completely.

Before booting up APs, the BSP should first collect information about the multiprocessor system, such as the total number of CPUs, their APIC IDs and the MMIO address of the LAPIC unit. This is done by functions defined in `kern/dev/pcpu_mp.c`. Don't be panic if you cannot understand some of the code in the file. Once you've skimmed through the file, further read `kern/pcpu/PCPUIntro/PCPUIntro.c`, `kern/pcpu/PCPUInit/PCPUInit.c`, and `kern/init/init.c` to try to understand the control flow transfer during the bootstrap of APs.

Per-CPU State and Initialization

When writing a multiprocessor OS, it is important to distinguish between per-CPU state that is private to each processor, and global state that the whole system shares.

Here is the per-CPU state you should be aware of:

- **Per-CPU information**. The `pcpu` structure defined in `kern/pcpu/PCPUIntro/PCPUIntro.c`.
- **Per-CPU kernel bootstrap stack**. Because multiple CPUs can trap into the kernel simultaneously, we need a separate kernel stack for each processor to prevent them from interfering with each other's execution. The array `struct kstack bsp_kstack[NUM_CPUS]` defined in `kern/clib/kstack.h` reserves space for NUM_CPUS's worth of kernel bootstrap stacks. Once the kernel is started, then each kernel thread uses its own stack, allocated in `struct kstack proc_kstack[NUM_IDS]` defined in the same file.
- **Per-CPU TSS and TSS descriptor**. A per-CPU task state segment (TSS) is also needed in order to specify where each CPU's kernel stack lives. You can refer to `kern/lib/seg.c` for how the TSS is initialized for each CPU.

- **Per-CPU current thread ID**. Since each CPU can run different user processes simultaneously, we need to remember the ID of currently running thread for each CPU. Read `kern/thread/PCurID/PCurID.c` for more details.

- **Per-CPU thread queues**. Using one thread ready queue would eliminate concurrency in the scheduler, since at any moment, at most one CPU's scheduler could access the ready queue. Instead, we allocate one ready queue for each scheduler on each processor. Check `kern/thread/PTQueueIntro/PTQueueIntro.c` for more details.

- **Per-CPU trap handlers**. Because multiple CPUs can trap into the kernel simultaneously, we need to register the appropriate trap handlers for each CPU. Part of the code is implemented in `kern/trap/TTrapInit/TTrapInit.c`.

> ## Exercise 2
>
> Complete the implementation of `trap_init` in `kern/trap/TTrapInit/TTrapInit.c` by registering appropriate trap handler for each trap number. You can refer to `kern/dev/intr.h` for the list of trap numbers.

Locking

We are now done with initializing the per-CPU states in the kernel. Before going any further, we need to first address race conditions when multiple CPUs run kernel code simultaneously. We would like to prevent the situation where multiple processes trying to write to global system-shared states simultaneouly. The simplest way to achieve this is to use a *big kernel lock*. The big kernel lock is a single global lock that is held whenever a process enters kernel mode, and is released when the process returns to user mode. In this model, processes in user mode can run concurrently on any available CPUs, but no more than one process can run in kernel mode; any other processes that try to enter kernel mode are forced to wait. The big kernel lock is simple and easy to use. Nevertheless, it eliminates all concurrency in kernel mode. Most modern operating systems use different locks to protect different parts of their shared states, an approach called fine-grained locking. Fine-grained locking can increase performance significantly, but is more difficult to implement and error-prone. mCertiKOS implements fine-grained locking. In this section, you will be asked to add different locks to various places in the kernel to ensure the conrrectness of the concurrent mCertiKOS kernel.

> ### Exercise 3
>
> This exercise is to help you get familiar with the locking implemented in mCertiKOS. You do not need to write any code for this exercise. We have already implemented a spinlock for you, which can be found in `kern/lib/spinlock.c`. In any of the subsequent exercises, you are free to use our implementation of spinlock wherever is convenient. We have also added locks to protect the physical memory allocators and the virtual memory allocators. Go through the relevant code to make sure you understand them.

> ### Exercise 4

> If you have not done it in assignment 1, read the following code carefully to understand their relationships: `kern/dev/serial.c`, `kern/dev/console.c`, `kern/lib/dprintf.c`, and `kern/lib/debug.c`. Once you have figured out how they work together, add appropriate locks to their implementations. You should try to use multiple locks to improve the performance, i.e., we do not want two non-related jobs waiting on the same lock. This is a warning that simply adding locks to every function WILL NOT WORK.

> **Exercise 5**
>
> Similarly, check out the scheduler implementation in `kern/thread/PThread/PThread.c`, and add locks in appropriate places.

This completes part 1. Compile and run your OS. Check the section on Test the User-Process Setup above to see the expected outcome.

Part 2: Preemptive Multitasking

In this part of the assignment, you will modify the kernel to preempt uncooperative processes.

Clock Interrupts and Preemption

As described in the section on Test the User-Process Setup, up until now, the first process on each CPU simply spins forever in a tight loop (see `user/lib/entry.S`) after the main function returns, and thus, the second process never gains the CPU. This is obviously not an ideal situation in terms of protecting the system from bugs or malicious code in user-mode processes, because any user-mode process can bring the whole system to a halt simply by getting into an infinite loop and never giving back the CPU. In order to allow the kernel to *preempt* a running process, forcefully retaking control of the CPU from it, the mCertiKOS kernel must support external hardware interrupts from the clock hardware.

Interrupt discipline

External interrupts (i.e., device interrupts) are referred to as IRQs. There are 16 possible IRQs, numbered 0 through 15. The mapping from IRQ number to IDT entry is not fixed. `intr_init_idt` in `kern/dev/intr.c` maps IRQs 0-15 to IDT entries `T_IRQ0` through `T_IRQ0+15`.

In `kern/dev/intr.h`, `T_IRQ0` is defined to be decimal 32. Thus the IDT entries 32-47 correspond to the IRQs 0-15. For example, the clock interrupt is IRQ 0. Thus, idt[T_IRQ0+0] (i.e., idt[32]) contains the address of the clock's interrupt handler routine in the kernel. This `T_IRQ0` is chosen so that the device interrupts do not overlap with the processor exceptions, which could obviously cause confusion. (In fact, in the early days of PCs running MS-DOS, the `T_IRQ0` effectively *was* zero, which indeed caused massive confusion between handling hardware interrupts and handling processor exceptions!)

In this part of the assignment, we make a key simplification. External device interrupts are *always* disabled when in the kernel but they are always enabled when in user space. We will mitigate this assumetion by allowing the external interrupts in some designated parts of the kernel in Part 3. External interrupts are controlled by the `FL_IF` flag bit of the `%eflags` register. When this bit is set, external interrupts are enabled. Once a user process traps into the kernel, we immediately disable the interrupt (check the `cli` command in first line of function `_alltraps` in `kern/dev/idt.S`, which is an x86 instruction that disables the interrupt.). Note that when we reach `_alltraps`, the CPU has already saved the old `%eflags` register in the current trap frame, pushed on to the current thread's stack, specified in the TSS. Thus, when we return back to the user with the `iret` instruction, the `%eflags` register was restored with the original value saved in the trap frame and thus the interrupts are turned back on when we reach the user space.

Handling Clock Interrupts

We need to program the hardware to generate clock interrupts periodically, which will force control back to the kernel where we can switch control to a different user process.

The calls to `lapic_init`, which we have written for you in `kern/dev/lapic.c`, sets up the clock and the interrupt controller to generate interrupts. You now need to write the code to handle these interrupts.

> ### Exercise 6
>
> Modify the function `timer_intr_handler` in `kern/trap/TTrapHandler/TTrapHandler.c` to switch to another thread in every `SCHED_SLICE` miliseconds, defined in `kern/lib/thread.h`. The constant `LAPIC_TIMER_INTR_FREQ` (`kern/dev/lapic.h`) defines the frequency of the LAPIC timer interrupt (# interrupts per second). You should first implement a function named `sched_update` in `kern/thread/PThread/PThread.c` that records (for each CPU) the number of miliseconds that has elapsed since the last thread switch. Once that reaches `SCHED_SLICE` miliseconds, you should yield the current thread to other ready thread by calling `thread_yield`, and reset the counter. Add locking statements as necessary. Finally, in `timer_intr_handler`, you can simply invoke this new function.

Now, check whether your current program execution meets the expectation described in the section on Test the User-Process Setup.

Part 3: Preempting Kernel Execution

Up till now, we already have a concurrent mCertiKOS with multicore and preemptive multitasking support. On the other hand, the interrupts are always turned off in the kernel. In particular, the kernel cannot be interrupted by the timer and thus any kernel execution cannot be preempted.

> ### Exercise 7
>
> Modify `syscall_dispatch` to temporarily enable interrupts during the executions of `sys_produce` and `sys_consume`. You will find `intr_local_enable` and `intr_local_disable` defined in `kern/dev/intr.h` helpful.

There are a couple of things that you need to consider when you turn on the interrupts in the kernel.

First, our old way of saving/restoring the trap frames (check the assignment 3) no longer works. In the last assignment, when a user program traps into the kernel, in the function `trap`, we save the current trap frame into memory (`uctx_pool`). From then on, whenever we need to read the values in the trap frame (to retrieve the system call number and function arguments), or write values to the trap frame (to set the error number and return values), we directly manipulate `uctx_pool`, instead of the actual trap frame pushed onto the kernel stack. Later when we return to the user, we restore all the user states from `uctx_pool` as well. Doing so can have some advantages. For example, the trap frame is saved into the memory instead of the stack, thus it is more secure, e.g., it may not be overwritten by unexpected stack overflow. Furthermore, the trap frame is saved into a global memory, which can be accessed by any part of the kernel code. Thus, you do not have to pass along the trap frame stack pointers everywhere, making the implementation more clean. Also, passing pointers to the stack allocated variables in C are generally considered dangerous.

However, this scheme no longer works as soon as you enable interrupts in any part of the kernel. Consider the scenario where the user called `produce` system call, and now we are in the function `trap`. Then we save the current trap frame into `uctx_pool`, then dispatch the request to the handler `sys_produce`. At this moment, you turned on the interrupt, so the kernel execution is interrupted by a timer interrupt. Then now you are back in the function `trap` again to handle this timer interrupt. Note that here we have nested interrupts, a timer interrupt nested inside a user system call. Now in this `trap` function, you save the current trap frame into `uctx_pool`... wait, you overwrote the existing trap frame for the outer system call!

Well, now you may think that this is easy to solve. You just need to define another `kictx_pool` and save the trap frame into this location if you have detected that the interrupt was triggered inside the kernel (assuming there's an easy way to detect this). This seems to work because in the current set up, the maximum depth of the nested interrupt calls is only two. This is because we always turn off the interrupt whenever there is a trap, and we only temporarily reenable it for the system call handlers of producer and consumer. Thus, when there is a timer interrupt in the kernel, it goes to the interrupt handler and this part of the code cannot be interrupted. Unfortunately, this solution does not work either. The simple reason is that it does not restore correct kernel stack pointer. Check the function `proc_start_user` in `kern/proc/PProc/PProc.c` on how we went back to the user from the kernel. We did it by calling `trap_return` to the saved trap frame in `uctx_pool`. Now check the implementation of `trap_return` in `kern/dev/idt.S`. Especially, check the first line in the function body. We are setting the current `%esp` register to the address of the argument (`uctx_pool` entry). Then we restore various registers for the user program by the `pop` instructions. As you may have already noticed, we are changing our kernel stack pointer to the address of one of the entries in `uctx_pool`. Previously, this was OK because nested interrupt was not possible, and every trap was triggered by the user. Thus, inside the trap handler, it is OK to "ruin" the kernel stack pointer, because we are about to return back to the user, and next time there is another trap from the user, the CPU will set the kernel stack pointer back to appropriate STACK_TOP configured in TSS! However, if an interrupt is triggered in the kernel, you definitely do not want to change the stack pointer to a random memory location.

Well, you may now think that this problem can be solved if we can somehow manage to set the `%esp` register back to the original value in the case that we are returnning back to the kernel instead of the user. This may work, but the implementation may not be trivial. Even worse, if we allow nested interrupts of bigger depth, e.g., if we turn on the interrupt in the interrupt handlers themselves, then you need to further turn `kictx_pool` into a linked list.

Instead of tackling all these complications, we have made a simple decision. We just use the original trap frame saved in the stack and pass the trap frame pointers along. If you check out the current implementation, you will notice that we only use `uctx_pool` for the process creation. In the function `trap`, we no longer save the current trap frame into the memory. Instead, we pass the trap frame pointer to all the trap handling functions, and those functions further pass the pointer to subsequent funtions like the ones retrive the system call arguments. After the trap is handled completely, then we simply call `trap_return` to the current trap frame on the stack. This way, after the trap returns, the stack will be the same as before. And it works perfectly with the nested interrupts, regardless of the nesting depth. With nested interrupts, the trap frames will just pile up onto the current stack and will eventually return back to the original stack as the interrupts getting handled.

So now we have solved the trap frame saving/restoring issue for you. But there will be another issue. Try to run your program, and you will most likely notice that your kernel gets hung in the middle of some calls to the print function. So what else could go wrong by turning on the interrupts for these two fixed places in the kernel? The issue was from the print calls inside `sys_produce` and `sys_consume`. Consider the case where you are in the middle of executing one of those print statements. At this moment, your thread holds the debuging lock used by the print statement. Then your execution is preempted, and the scheduler schedules another thread. That thread may be executing something other than producer or consumer system call handling code (thus the interrupt is off) and it may also want to print something to the screen. Since the debuging spinlock was held by some other thread, and the interrupt was turned off, the current thread will spin forever hoping that some other thread would release the lock, but the thread who holds the lock will not get any chance to run unless the current thread gives up the CPU.

The main problem here is that we intended to only turn on the interrupt within those two system call handlers. Therefore, the rest of the kernel modules (including the debugging modules) were written under the assumption that the interrupts were always turned off. Thus, the obvious solution is to turn off the interrupt temporarily whenever `sys_produce` or `sys_consume` tries to call any other kernel function, and to turn the interrupt back on once the function returns.

> ### Exercise 8
>
> Modify `sys_produce` and `sys_consume` to temporarily disable interrupts whenever it calls any printing function, and turn interrupts back on afterwards.

At this stage, you should be able to run the program and see the desired outputs (see the section on Test the User-Process Setup). There is still one simple (but very useful) optimization to make. Currently in the function `trap`, we always switch back the kernel stack and the page structure to the current thread's kernal stack and page structure. This is totally unnecessary in the case where the interrupt was triggered inside the kernel, because the current running thread is never changed. You may wonder why setting some old values to themselves can be considered harmful. These two simple statements have hidden performance impacts. Whenever you switches the TSS or the page structure, there's a TLB flush, and the hardware is not smart enough to figure out you are actually switching one to itself. Thus we want to avoid doing so if possible.

> ### Exercise 9
>
> Improve the implementation of `trap` by remembering the last active thread ID for each CPU, and only switch the TSS and page table when the saved ID is different from the current thread we are returning back to. You can change whatever files that makes implementing this improvement convenient.

Last, here is one more left-over exercise from the last assignment.

> ### Exercise 10
>
> An operating system should guarantee that under all possible arguments, call to any system calls do not go wrong. Thus the system call handlers are responsible to do all the argument validity checkings and return appropriate error code if any of the check fails. Currently in mCertiKOS, it is still possible that the call to `sys_spawn` goes wrong. I have added three more error code in `kern/lib/syscall.h`, mainly `E_EXCEEDS_QUOTA`, `E_MAX_NUM_CHILDEN_REACHED`, and `E_INVAL_CHILD_ID`. They should be self explanatory from their names. You are asked to enhance the implementation of `sys_spawn` to do all the argument checks in the body to detect possible errors and set appropriate error codes, making sure any calls to `sys_spawn`, under all possible arguments, never go wrong. You should be able to do so by just changing the implementation of `sys_spawn`.

Part 4: The Producer-Consumer Problem

From Wikipedia:

> In computing, the producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

In this part of the assignment, you are asked to implement one particular version of the producer-consumer problem. We have implemented `produce` and `consume` as system calls and created multiple user processes to invoke these system calls. Recall that we have created two producer user processes in CPU #1 and two consumer user processes in CPU #2 (check `kern/init/init.c`). When the buffer is full, the process who called `produce` system call should be put into a waiting list, later woken up by consumers when the buffer becomes not full. Similar blocking mechanism should be implemented for the `consumer` as well, i.e., the consumers should be blocked if the buffer is empty. You should implement it using the shared objects and condition variables. You can find more information on how to implement bounded buffer in the Chapter 5 of the textbook.

> ### Exercise 11
>
> Implement appropriate condition variables, and implement a bounded buffer as shared object. You can use the provided spinlock implemention in `kern/lib/spinlock.c`. Then modify the `sys_produce` and `sys_consume`

> system call handlers to manipulate the bounded buffer accordingly. Delete the existing code in those two functions. Change `user/include/syscall.h` and `user/lib/proc.c` accordingly if necessary. Feel free to change the signatures of relavent functions and define new functions as needed. Do not try to copy exact code from the book. It will not work as it is, because we are in a slightly different setting, e.g., each CPU has its own ready queue in mCertiKOS. You should add appropriate (easy to understand) debuging outputs (either in the system call handlers, or in the user program, or in both) to illustrate that your implementation actually works, with all the settings you've implemented in the Part 1-3. Explain your implementation in the README file.

This completes the lab. Don't forget editing your `README` file. Add each of the files that you created to the git repository with the command `git add /path/to/the/new/file`. After this, you should not see any of the files you want to submit under the "Untracked files" section when you run `git status`. Commit your changes and type `git push` in the `mcertikos` directory to submit your code.